

PATENT ABSTRACTS OF JAPAN

(11)Publication number : 2001-216161

(43)Date of publication of application : 10.08.2001

(51)Int.Cl.

G06F 9/42

(21)Application number : 2000-028358

(71)Applicant : INTERNATL BUSINESS MACH CORP <IBM>

(22)Date of filing : 04.02.2000

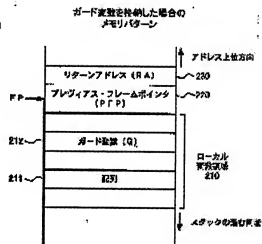
(72)Inventor : ETO HIROAKI
YODA KUNIKAZU

(54) MEMORY DEVICE, STACK PROTECTION SYSTEM, COMPUTER SYSTEM, COMPILER, STACK PROTECTING METHOD, STORAGE MEDIUM AND PROGRAM TRANSMITTER

(57)Abstract:

PROBLEM TO BE SOLVED: To prevent stack smashing attack due to buffer overflow on a stack.

SOLUTION: This memory device is used by a computer system. It is provided with an area to store a return address 230 to a call origin of a function under execution, an area to store a previous frame pointer 22 to the call origin of the function under execution and an area to be located in the rear of the area to store the return address 230 and the area to store the previous frame pointer 220 and to store local variables as memory patterns of this memory device after invoking the function when this computer system executes programs. The area to store the local variables stores a guard variable 212 as an object to confirm whether or not it is destroyed in a return processing of the function under execution before this arrangement 211 when the arrangement 211 is stored in the area to store the local variables.



LEGAL STATUS

[Date of request for examination] 29.09.2000

[Date of sending the examiner's decision of rejection]

[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]

[Date of final disposal for application]

[Patent number] 3552627

[Date of registration] 14.05.2004

[Number of appeal against examiner's decision of rejection]

[Date of requesting appeal against examiner's decision of rejection]

[Date of extinction of right]

(51) Int.Cl.⁷

G 0 6 F 9/42

識別記号

3 3 0

F I

G 0 6 F 9/42

データ種別 (参考)

3 3 0 C 5 B 0 3 3

審査請求 有 請求項の数19 O L (全 15 頁)

(21) 出願番号 特願2000-28358 (P2000-28358)

(22) 出願日 平成12年2月4日 (2000.2.4)

(71) 出願人 390009531

インターナショナル・ビジネス・マシーン
ズ・コーポレーションINTERNATIONAL BUSIN
ESS MACHINES CORPO
RATION

アメリカ合衆国10504、ニューヨーク州

アーモンク (番地なし)

(74) 復代理人 100104880

弁理士 古部 次郎 (外3名)

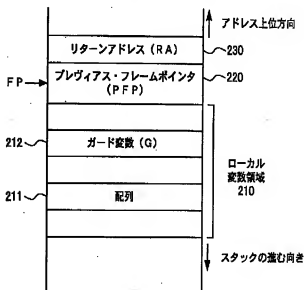
最終頁に続く

(54) 【発明の名称】 メモリ装置、スタック保護システム、コンピュータシステム、コンパイラ、スタック保護方法、
記憶媒体及びプログラム伝送装置

(57) 【要約】

【課題】 スタック上のバッファオーバーフローを原因とするスタックスマッシング攻撃を防止する。

【解決手段】 コンピュータシステムが使用するメモリ装置であって、このコンピュータシステムがプログラムを実行する際の関数呼び出し後におけるこのメモリ装置のメモリパターンとして、実行中の関数の呼び出し元へのリターンアドレス230を格納する領域と、この実行中の関数の呼び出し元へのプレヴィアス・フレームポイント22を格納する領域と、このリターンアドレス230を格納する領域とこのプレヴィアス・フレームポイント220を格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域は、このローカル変数を格納する領域に配列211が格納されている場合に、この配列211よりも前に、実行中の関数のリターン処理において破壊されないかどうかを確認する対象であるガード変数212を格納する。

ガード変数を格納した場合の
メモリパターン

【特許請求の範囲】

【請求項1】 コンピュータシステムが使用するメモリ装置であって、

前記コンピュータシステムがプログラムを実行する際の関数呼び出し後における前記メモリ装置のメモリパターンとして、

実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、

前記実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、

前記リターンアドレスを格納する領域と前記プレヴィアス・フレームポインタを格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、

前記ローカル変数を格納する領域は、当該ローカル変数を格納する領域に配列が格納されている場合に、当該配列よりも前に、前記実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したメモリパターンを有することを特徴とするメモリ装置。

【請求項2】 前記メモリ装置の前記メモリパターンにおける前記ローカル変数を格納する領域は、当該ローカル変数を格納する領域に文字配列が格納されている場合に、当該配列よりも前に、前記ガード変数を格納したことを特徴とする請求項1に記載のメモリ装置。

【請求項3】 前記メモリ装置の前記メモリパターンにおける前記ローカル変数を格納する領域に格納される前記ガード変数は乱数であることを特徴とする請求項1に記載のメモリ装置。

【請求項4】 コンピュータシステムが使用するメモリ装置であって、

前記コンピュータシステムがプログラムを実行する際の関数呼び出し後における前記メモリ装置のメモリパターンとして、

実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、

前記リターンアドレスを格納する領域の後方に位置し、前記実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、

前記プレヴィアス・フレームポインタを格納する領域の後方に位置し、ローカル変数を格納する領域とを備え、前記ローカル変数を格納する領域に配列が格納されている場合に、前記プレヴィアス・フレームポインタと当該配列との間に、前記実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したメモリパターンを有することを特徴とするメモリ装置。

【請求項5】 コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護システムであって、ソース形式のプログラムを入力し、関数呼び出し後のス

タックにおけるプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される配列との間にガード変数を格納する指令を当該ソース形式のプログラムに付加するスタック保護指令作成部と、

前記スタック保護指令作成部により前記ガード変数を格納する指令を付加されたプログラムを実行し、当該ガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに当該ガード変数を格納すると共に、当該関数のリターン処理において当該ガード変数の有効性を確認するスタック保護処理実行部とを備えることを特徴とするスタック保護システム。

【請求項6】 前記スタック保護処理実行部は、前記関数のリターン処理において前記ガード変数が破壊されていることを検出した場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴とする請求項5に記載のスタック保護システム。

【請求項7】 前記スタック保護指令作成部は、コンパイラに実装され、前記コンパイラが、前記ソース形式のプログラムとしてコンパイラ言語にて記述されたソースプログラムを入力し、当該ソースプログラムをオブジェクトプログラムに翻訳する際に、当該オブジェクトプログラムに対して、前記ガード変数を格納する指令を付加することを特徴とする請求項5に記載のスタック保護システム。

【請求項8】 コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護システムであって、前記関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを入力して実行すると共に、当該ガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに前記ガード変数を格納し、当該関数のリターン処理において当該ガード変数の有効性を確認するプログラム実行手段と、

前記プログラム実行手段により、前記関数のリターン処理において前記ガード変数が破壊されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行う異常終了実行手段とを備えることを特徴とするスタック保護システム。

【請求項9】 種々の演算処理を行うデータ処理装置と、当該データ処理装置が演算処理を行う際に用いるメモリ装置とを備えたコンピュータシステムであって、前記データ処理装置が演算処理を実行する際の関数呼び出し後における前記メモリ装置のメモリパターンは、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、前記実行中の関数の呼び出し元へのプレヴィアス・フレ

ームポインタを格納する領域と、
前記リターンアドレスを格納する領域と前記プレヴィアス・フレームポインタを格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、
前記ローカル変数を格納する領域は、当該ローカル変数を格納する領域に配列が格納されている場合に、当該配列よりも前に、前記実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したことを特徴とするコンピュータシステム。

【請求項 10】 種々の演算処理を行うデータ処理装置と、当該データ処理装置が演算処理を行う際に用いるメモリ装置とを備えたコンピュータシステムであって、前記データ処理装置が演算処理を実行する際の関数呼び出し後における前記メモリ装置のメモリパターンは、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、

前記リターンアドレスを格納する領域の後方に位置し、前記実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、
前記プレヴィアス・フレームポインタを格納する領域の後方に位置し、ローカル変数を格納する領域とを備え、
前記ローカル変数を格納する領域に配列が格納されている場合に、前記プレヴィアス・フレームポインタと当該配列との間に、前記実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したことを特徴とするコンピュータシステム。

【請求項 11】 プログラム制御により種々の演算処理を行うコンピュータシステムであって、
関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを読み込んで実行するデータ処理装置と、

前記データ処理装置が演算処理を行う際に用いるメモリ装置とを備え、

前記データ処理装置は、前記ガード変数を格納する指令が付加されたプログラムにおける前記ガード変数を格納する指令にしたがって、関数呼び出しの際に、前記メモリ装置上のスタックに当該ガード変数を格納すると共に、当該関数のリターン処理において当該ガード変数の有効性を確認することを特徴とするコンピュータシステム。

【請求項 12】 前記データ処理装置は、
前記関数のリターン処理において前記ガード変数が破壊されていることを検出した場合に、前記プログラムの実行を中止し、スタックマッピング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴とする請求項 11 に記載のコンピュータシステム。

【請求項 13】 プログラム制御により種々の演算処理を行うコンピュータシステムであって、
ソース形式のプログラムを入力し、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される配列との間にガード変数を格納する指令を当該ソース形式のプログラムに付加すると共に、当該ガード変数を格納する指令を付加されたプログラムを実行するデータ処理装置と、
前記データ処理装置が演算処理を行う際に用いるメモリ装置とを備え、

前記データ処理装置は、前記ガード変数を格納する指令を付加されたプログラムにおける前記ガード変数を格納する指令にしたがって、関数呼び出しの際に、前記メモリ装置上のスタックに当該ガード変数を格納すると共に、当該関数のリターン処理において当該ガード変数の有効性を確認することを特徴とするコンピュータシステム。

【請求項 14】 前記データ処理装置は、
前記関数のリターン処理において前記ガード変数が破壊されていることを検出した場合に、前記プログラムの実行を中止し、スタックマッピング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴とする請求項 13 に記載のコンピュータシステム。

【請求項 15】 ソースプログラムを入力しオブジェクトプログラムに翻訳して出力するコンパイラであって、プログラムを翻訳する翻訳手段と、
プログラムの翻訳の際に、サブルーチンごとに、当該サブルーチンに含まれる関数が配列を持っているかどうかを調べ、配列を持っている場合に、当該サブルーチンに対して、当該関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される当該配列との間にガード変数を格納すると共に、関数のリターン処理において当該ガード変数の有効性を確認する旨の指令を付加するスタック保護指令付加手段とを備えることを特徴とするコンパイラ。

【請求項 16】 コンピュータによるプログラムの実行の際にスタックマッピング攻撃からプログラムカウンタを保護するスタック保護方法であって、
ソース形式のプログラムに対して、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される配列との間にガード変数を格納する指令を当該ソース形式のプログラムに付加するステップと、
前記ガード変数を格納する指令を付加されたプログラムを実行すると共に、当該ガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに当該ガード変数を格納し、関数のリターン処理において当該ガード変数の有効性を確認するステップと、
前記関数のリターン処理において前記ガード変数が破壊

されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知するステップとを含むことを特徴とするスタック保護方法。

【請求項 17】 コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護方法であって、前記関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを実行すると共に、当該ガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに当該ガード変数を格納し、関数のリターン処理において当該ガード変数の有効性を確認するステップと、前記関数のリターン処理において前記ガード変数が破壊されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知するステップとを含むことを特徴とするスタック保護方法。

【請求項 18】 コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能に記憶した記憶媒体において、前記プログラムは、配列を持っている関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される当該配列との間にガード変数を格納する処理と、前記関数のリターン処理において前記ガード変数の有効性を確認する処理と、前記関数のリターン処理において前記ガード変数が破壊されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する処理とを前記コンピュータに実行させることを特徴とする記憶媒体。

【請求項 19】 コンピュータに、配列を持っている関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される当該配列との間にガード変数を格納する処理と、前記関数のリターン処理において当該ガード変数の有効性を確認する処理と、前記関数のリターン処理において前記ガード変数が破壊されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する処理とを実行させるプログラムを記憶する記憶手段と、前記記憶手段から前記プログラムを読み出して当該プログラムを送信する送信手段とを備えたことを特徴とするプログラム伝送装置。

【発明の属する技術分野】本発明は、スタックスマッシング攻撃によるコンピュータへの侵入からプログラムカウンタを保護するスタック保護方法に関する。

【0002】

【従来の技術】コンピュータシステムへの侵入は、一般に、システム設定の不備やセキュリティホールと呼ばれるシステムの弱点を攻撃して他人の権限で侵入し、次の攻撃でシステム管理者権限を得るといった手順で行われる。

【0003】このような攻撃の典型的なものとしてスタックスマッシング攻撃がある。スタックには、プログラム内のサブルーチンの実行状態を保持するために、サブルーチンのリターンアドレス、ローカル変数、引数などが保管される。そして、スタックスマッシング攻撃は、スタック内のリターンアドレスを書き換えることによって悪意のあるプログラムに実行制御を移すという手法である。

【0004】スタックの構造を示してスタックスマッシング攻撃について具体的に説明する。図9はC言語における関数呼び出し後のスタック状態を示している。図9を参照すると、スタックトップ（図では下側）から順に、ローカル変数（Local Variable: LV）、当該関数の呼び出し元である直前の関数のフレームポインタ（Previous frame pointer: PFP）、及び当該呼び出し元へのリターンアドレス（Return Address: RA）が積まれている。ここで、フレームポインタ（frame pointer: FP）とは、ローカル変数が相対参照される際のベースポインタの事である。スタックスマッシング攻撃においては、ローカル変数（特に配列）への代入操作を行い、境界チェックが不完全な場合に、そのローカル変数より上位のスタック領域を破壊する。そして、スタック領域を破壊する際に、プログラムを書き込むと共に、リターンアドレスの領域にそのプログラムを指すような値を書き込む。この状態で関数のリターン処理が行われると、書き込まれたプログラムが起動することとなる。

【0005】C言語で記述されたプログラムを例にして、スタックスマッシング攻撃について、さらに具体的に説明する。図10はスタックスマッシング攻撃の攻撃対象であるC言語で記述されたプログラムの例であり、図11はこのプログラムに対してスタックスマッシング攻撃が行われた場合のスタックの状態を説明する図である。図10において、関数fooは環境変数HOMEの内容をローカル変数にコピーするプログラムである。関数strcpyはローカル変数の大きさを考慮しないため、環境変数HOMEの内容が128文字以上である場合、図11のbufから上方向の内容（網掛け箇所）を壊すこととなる。この破壊内容にプログラムの埋め込み、かつリターンアドレスの領域にそのプログラムの先頭アドレスを置くことによって、埋め込んだプログラムへ制御を移すことができる。

【0006】上述したスタックマッシング攻撃などにより、他人の管理するコンピュータネットワークに侵入する行為は、それ自身が機密漏洩やシステム破壊などを引き起こす場合がある。また、他人のIDやパスワードを盗み、その正当な所有者のふりをしてシステムに侵入するなりすまし行為や、他のコンピュータに侵入するための足がかりとして利用される場合もある。従来、この種のスタックマッシング攻撃は、UNIXサーバを主な攻撃目標としていた。しかし最近では、ActiveX (Acrobat Control for ActiveXなど (なお、ActiveXは米国マイクロソフト・コーポレーションの商標) のバッファオーバーフローが問題となり、これを対象とする攻撃が行われるようになっていた。そのため、ネットワークに接続するコンピュータ一般、すなわち、サーバマシン、クライアントマシン、PDAなどの全てが攻撃目標となり得る。バッファオーバーフローを起こすバグは、発見されてから取り除かれるというのが一般的である。したがって、このバグに対する攻撃を防ぐには、そのような攻撃に対する何らかの予防措置をとることが有効であると考えられる。そこで、これらのコンピュータシステムへの侵入を防止するために、従来から種々の方策が提案されている。

【0007】この種の従来技術の例としては、例えば、文獻「Automatic Detection and Prevention of Buffer-Overflow Attacks」(Crispin Cowan, Calton Pu, David Majer, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, in the 7th USENIX Security Symposium, San Antonio, TX, January 1998.)に開示されたStackGuardという技術がある。StackGuardでは、図9に示したようなスタック状態におけるリターンアドレスの領域と呼び出し元のフレームポインタの領域との間に、リターンアドレスの保護のための特別な値であるguard_valueを挿入する。そして、関数のリターン処理において、guard_valueの有効性を確認することにより、攻撃の有無を確認する。

【0008】上記従来技術の他にも、仮想記憶のページを操作し、スタック上のリターンアドレス格納箇所を書き込み禁止にする手法や、リターンアドレス専用のスタックを別に作成し、関数の入口でリターンアドレスを専用スタックにコピーし、関数リターン処理で専用スタックから戻すことにより、リターンアドレスを保護する手法などが提案されている。また、バッファオーバーフローを原因とするあらゆる攻撃を防止するために、全ての配列アクセスに対して境界チェックを行うという手法も考えられる。

【0009】

【発明が解決しようとする課題】しかし、上述したコンピュータシステムへの侵入を防止するための従来技術は、次のような欠点を有している。上記文獻に記載されたStackGuardは、実行中の関数の呼び出し元である直前

の関数へのリターンアドレスと当該直前の関数におけるフレームポインタ (以下、PFPと略す) との間に、guard_valueを格納する。このため、リターンアドレスは保護されているが、変数スコープを指し示すフレームポインタは保護されていない。これにより、重大なセキュリティホールが存在している。すなわち、フレームポインタを保護していないために、プログラムカウンタを操作することが可能である。以下に、その概要を説明する。

【0010】関数リターン処理では、次の各処理が行われる。

```
FP      ←      SP
PFP     ←      FP
(SP)    ←      PC      ; return operation
```

ここで、FPはフレームポインタ、SPはスタックポインタ、PCはプログラムカウンタである。また、矢印は値を代入することを示し、括弧は間接参照することを示す。上記のように、StackGuardでは、スタック内のPFPを保護していない。そのため、外部からの攻撃により、PFPの値を操作することが可能である。値の操作について、その依頼を調べると、上記の関数リターン処理を参照すれば、PFPを操作することにより一度のリターンでフレームポインタを操作することができる。そして、二度目のリターンでプログラムカウンタを操作することができる。実際に悪意のあるコードを埋め込むには、guard_valueテストを通過させなくてはならないが、StackGuardがデフォルトで使用しているguard_valueに対してはスタックマッシング攻撃が可能である。そしてさらに、その他のguard_valueについても攻撃手法が見つかる可能性がある。したがって、StackGuardで用いられる従来の手法では、スタックマッシング攻撃を完全に防止することができなかった。

【0011】また、StackGuardでは、一度目のリターンで汚されたフレームポインタの影響を検出するために、二度目のリターンでは必ずguard_valueの有効性を確認しなくてはならない。そのため、全ての関数において、guard_valueの有効性を確認するプログラムを発生させている。したがって、guard_valueの有効性を確認する処理のために、必ずある程度のオーバーヘッドが発生していた。さらにまた、StackGuardにおいては、リターンアドレス領域とPFP領域との間にguard_valueを挿入するため、スタックのフレーム構造を変更することとなる。このため、StackGuardを実装したプログラムに対してデバッグ支援を行うことができなかった。

【0012】仮想記憶のページを操作し、スタック上のリターンアドレス格納箇所を書き込み禁止にする従来技術においては、リターンアドレスのみを保護し、フレームポインタを保護していないため、StackGuardと同様のセキュリティホールが存在し、スタックマッシング攻撃を完全に防止することができなかった。また、スタッ

クページへの書き込みを行うときにはスーパーバイザーモードに制御が移るため、実行オーバーヘッドが大きいという欠点があった。

【0013】リターンアドレス専用のスタックを別に作成し、関数の入口でリターンアドレスを専用スタックにコピーし、関数リターン処理で専用スタックから戻す従来技術においては、リターンアドレスへの直接的な攻撃が可能であった。すなわち、グローバル変数領域に作成された専用スタックは、その値の有効性を確認しないで元のスタックに戻すため、専用スタックからリターンアドレスを戻す操作への攻撃が直接リターンアドレスへの攻撃となってしまう。また、同従来技術は、専用スタックの管理オーバーヘッドが大きいという欠点があった。

【0014】全ての配列アクセスに対して境界チェックを行う場合は、上述したようにバッファオーバーフローの原因とするあらゆる攻撃を防止することができるが、主にポインタを使用したプログラムではオーバーヘッドが非常に大きいという欠点があった。

【0015】本発明は以上のような技術的課題を解決するためになされたものであって、スタック上のバッファオーバーフローを原因とするスタックスマッシング攻撃を完全に防止することを目的とする。また、スタックスマッシング攻撃の防止手段を実装したことによる実行オーバーヘッドを削減することを他の目的とする。

【0016】

【課題を解決するための手段】かかる目的のもと、本発明は、コンピュータシステムが使用するメモリ装置であって、このコンピュータシステムがプログラムを実行する際の関数呼び出し後におけるこのメモリ装置のメモリパターンとして、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、この実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、このプレヴィアス・フレームポインタを格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域は、このローカル変数を格納する領域に配列が格納されている場合に、この配列よりも前に、実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したメモリパターンを有することを特徴としている。このような構成とすれば、リターンアドレスと共にプレヴィアス・フレームポインタをも保護することにより、スタックスマッシング攻撃を用いたシステムへの侵入を確実に阻止することができる。さらに、ガード変数を用いたスタック保護処理の対象を、ローカル変数を格納する領域に配列が格納されている場合に限定することにより、システム全体のオーバーヘッドを低減させることができる。

【0017】ここで、メモリ装置のメモリパターンにおけるローカル変数を格納する領域は、このローカル変数

を格納する領域に文字配列が格納されている場合に、この配列よりも前に、ガード変数を格納する構成とすることができる。すなわち、ローカル変数を格納する領域に、配列のうちの文字配列が格納されている場合にのみ、ガード変数を用いたスタック保護処理を実行する。このような構成とすれば、文字配列を持たない関数の他に整数配列のような文字配列以外の配列を持つ関数に対してもガード変数を用いたスタック保護処理を行わないため、システム全体のオーバーヘッドを更に低減させることができる。スタックスマッシング攻撃の対象となる配列は、主に文字配列であるため、ガード変数を用いたスタック保護処理の対象を文字配列を持つ関数のみに限定しても、安全性が大幅に損なわれることはない。そこで、処理速度が重視されるシステムでは、このような構成とすることが好ましい。

【0018】また、ローカル変数を格納する領域に格納されるガード変数として、乱数を用いる構成とすることができる。このほか、ガード変数としては、攻撃者が知ることのできない任意の値を用いることができる。

【0019】さらに本発明においては、コンピュータシステムが使用するメモリ装置であって、このコンピュータシステムがプログラムを実行する際の関数呼び出し後におけるこのメモリ装置のメモリパターンとして、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、このリターンアドレスを格納する領域の後方に位置し、実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、このプレヴィアス・フレームポインタを格納する領域の後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域に配列が格納されている場合に、このプレヴィアス・フレームポインタとこの配列との間に、実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したメモリパターンを有することを特徴としている。すなわち、ガード変数を格納する位置は、スタック上のプレヴィアス・フレームポインタと配列との間であれば、どこでも良い。

【0020】また、本発明は、コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護システムであって、ソース形式のプログラムを入力し、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する指令をこのソース形式のプログラムに付加するスタック保護指令作成部と、このスタック保護指令作成部によりガード変数を格納する指令を付加されたプログラムを実行し、このガード変数を格納する指令にしたがって、関数呼び出しの際にスタックにこのガード変数を格納すると共に、この関数のリターン処理においてこのガード変数の有効性を確認するスタック

ク保護処理実行部とを備えることを特徴としている。このような構成とすれば、プログラムを実行する際に、リターンアドレスと共にプレヴィアス・フレームポインタをも保護することができ、スタックスマッシング攻撃を用いたシステムへの侵入を確実に阻止することができる点で優れている。

【0021】ここで、スタック保護処理実行部は、関数のリターン処理においてガード変数が破壊されていることを検出した場合に、プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行う構成とすることができる。このような構成とすれば、スタックスマッシング攻撃が行われた場合には、かかる関数の呼び出し元へ処理が戻る直前でプログラムの実行を中止するため、システムへの侵入を効果的に防止することができる点で好ましい。

【0022】また、スタック保護指令作成部は、コンパイルに実装され、このコンパイラが、ソース形式のプログラムとしてコンパイル言語にて記述されたソースプログラムを入力し、このソースプログラムをオブジェクトプログラムに翻訳する際に、このオブジェクトプログラムに対して、ガード変数を格納する指令を付加する構成とすることができる。このような構成とすれば、オーバーフローを引き起こす変数よりもスタック上で上位アドレスにガード変数の宣言を配置すること、及び全ての関数の出口にプログラムを挿入することが容易となる点で好ましい。

【0023】さらに本発明においては、コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護システムであって、関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを入力して実行すると共に、このガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに前記ガード変数を格納し、この関数のリターン処理においてこのガード変数の有効性を確認するプログラム実行手段と、このプログラム実行手段により、関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行う異常終了実行手段とを備えることを特徴としている。このような構成とすれば、手作業を含むどのような手段でプログラムにガード変数を格納する指令を付加した場合であっても、そのようなプログラムを実行する際にスタック保護を実行できる点で好ましい。

【0024】また、本発明は、種々の演算処理を行うデータ処理装置と、このデータ処理装置が演算処理を行う際に用いるメモリ装置とを備えたコンピュータシステムであって、このデータ処理装置が演算処理を実行する際

の関数呼び出し後におけるこのメモリ装置のメモリバターンは、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、このリターンアドレスを格納する領域とこのプレヴィアス・フレームポインタを格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域は、このローカル変数を格納する領域に配列が格納されている場合に、この配列よりも前に、実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したことを特徴としている。このような構成とすれば、リターンアドレスと共にプレヴィアス・フレームポインタをも保護することにより、スタックスマッシング攻撃を用いたシステムへの侵入を確実に阻止することができる。さらに、ガード変数を用いたスタック保護処理の対象を、ローカル変数を格納する領域に配列が格納されている場合に限定することにより、システム全体のオーバーヘッドを低減させることができる。

【0025】また、本発明は、種々の演算処理を行うデータ処理装置と、このデータ処理装置が演算処理を行う際に用いるメモリ装置とを備えたコンピュータシステムであって、このデータ処理装置が演算処理を実行する際の関数呼び出し後におけるこのメモリ装置のメモリバターンは、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、このリターンアドレスを格納する領域の後方に位置し、実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、このプレヴィアス・フレームポインタを格納する領域の後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域に配列が格納されている場合に、このプレヴィアス・フレームポインタとこの配列との間に、実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したことを特徴としている。ガード変数を格納する位置は、スタック上のプレヴィアス・フレームポインタと配列との間であれば、どこでも良い。

【0026】さらにまた、本発明は、プログラム制御により種々の演算処理を行うコンピュータシステムであって、関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを読み込んで実行するデータ処理装置と、このデータ処理装置が演算処理を行う際に用いるメモリ装置とを備え、このデータ処理装置は、前記ガード変数を格納する指令が付加されたプログラムにおけるガード変数を格納する指令にしたがって、関数呼び出しの際に、メモリ装置上のスタックにこのガード変数を格納すると共に、この関数のリターン処理においてこのガード変数の有効性を確認することを特徴としている。こ

のような構成とすれば、ガード変数を格納する指令が付加されたプログラムを、磁気ディスクや光ディスクなどの種々の記憶媒体に記録して提供したり、ネットワークを介して提供したりすることにより、このプログラムをインストールしたコンピュータシステムにおいて、スタック保護を実行できる点で好ましい。

【0027】ここで、データ処理装置は、関数のリターン処理においてガード変数が破壊されていることを検出した場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴としている。このような構成とすれば、スタックスマッシング攻撃が行われた場合には、かかる関数の呼び出し元へ処理が戻る直前でプログラムの実行を中止するため、システムへの侵入を効果的に防止することができる点で好ましい。

【0028】また、本発明は、プログラム制御により種々の演算処理を行うコンピュータシステムであって、ソース形式のプログラムを入力し、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する指令をこのソース形式のプログラムに付加すると共に、このガード変数を格納する指令を付加されたプログラムを実行するデータ処理装置と、このデータ処理装置が演算処理を行う際に行うメモリ装置とを備え、このデータ処理装置は、このガード変数を格納する指令を付加されたプログラムにおけるガード変数を格納する指令にしたがって、関数呼び出しの際に、このメモリ装置上のスタックにこのガード変数を格納すると共に、この関数のリターン処理においてこのガード変数の有効性を確認することを特徴としている。このような構成とすれば、任意のプログラムに対して、ガード変数を格納する指令を付加し、このプログラムを実行することによって、スタックスマッシング攻撃を用いたシステムへの侵入を確実に阻止することができる点で優れている。ここで、プログラムは、コンパイラ言語で記述されたものであっても、インタプリタ言語で記述されたものであってもかまわない。コンパイラ言語で記述されたプログラムである場合は、データ処理装置は、コンパイラとしての機能も併せて有することとなる。

【0029】ここで、データ処理装置は、関数のリターン処理においてガード変数が破壊されていることを検出した場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴としている。このような構成とすれば、スタックスマッシング攻撃が行われた場合には、かかる関数の呼び出し元へ処理が戻る直前でプログラムの実行を中止するため、システムへの侵入を効果的に防止することができる点で好ましい。

【0030】また、本発明は、ソースプログラムを入力しオブジェクトプログラムに翻訳して出力するコンパイ

ラであって、プログラムを翻訳する翻訳手段と、プログラムの翻訳の際に、サブルーチンごとに、このサブルーチンに含まれる関数が配列を持っているかどうかを調べ、配列を持っている場合に、このサブルーチンに対して、この関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納すると共に、関数のリターン処理においてこのガード変数の有効性を確認する旨の指令を付加するスタック保護指令付加手段とを備えることを特徴としている。このような構成とすれば、ソースプログラムをコンパイルした時点で必要な指令をプログラムに付加することができる点で優れている。

【0031】さらにまた、本発明は、コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護方法であって、ソース形式のプログラムに対して、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する指令をこのソース形式のプログラムに付加するステップと、このガード変数を格納する指令を付加されたプログラムを実行すると共に、このガード変数を格納する指令にしたがって、関数呼び出しの際にスタックにこのガード変数を格納し、関数のリターン処理においてこのガード変数の有効性を確認するステップと、この関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知するステップとを含むことを特徴としている。

【0032】同様にして、コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護方法であって、関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令を付加されたプログラムを実行すると共に、このガード変数を格納する指令にしたがって、関数呼び出しの際にスタックにこのガード変数を格納し、関数のリターン処理においてこのガード変数の有効性を確認するステップと、関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知するステップとを含む構成とすることができる。このような構成とすれば、手作業を含むどのような手段でプログラムにガード変数を格納する指令を付加した場合であっても、そのようなプログラムを実行する際にスタック保護を実行できる点で好ましい。

【0033】また、本発明は、コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能

に記憶した記憶媒体において、このプログラムは、配列を持っている関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する処理と、関数のリターン処理においてこのガード変数の有効性を確認する処理と、関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する処理とをこのコンピュータに実行させることを特徴としている。このような構成とすれば、このプログラムを実行する全てのシステムにおいて、スタックスマッシング攻撃からプログラムカウンタを保護できる点で好ましい。なお、このプログラムは、その種類において何ら制限はなく、コンパイル言語であってもインタプリタ言語であっても良い。

【0034】さらにまた、本発明は、コンピュータに、配列を持っている関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する処理と、関数のリターン処理においてこのガード変数の有効性を確認する処理と、関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する処理とを実行させるプログラムを記憶する記憶手段と、この記憶手段からこのプログラムを読み出してこのプログラムを送信する送信手段とを備えたことを特徴としている。このような構成とすれば、このプログラム伝送装置からこのプログラムをダウンロードしてインストール可能なあらゆるシステムにおいて、スタックスマッシング攻撃からプログラムカウンタを保護できる。

【0035】

【発明の実施の形態】以下、添付図面に示す実施の形態に基づいてこの発明を詳細に説明する。まず、本発明の概要を説明する。本発明では、プログラムを実行する際のメモリパターン（記憶域の割当て）において、スタック上のプレヴィアス・フレームポインタ（Previous frame pointer：PFP）とローカル変数領域における配列との間に、スタックスマッシング攻撃からプログラムカウンタを保護するための特別な値であるガード変数を挿入する。そして、関数リターン処理において、ガード変数の有効性を確認することにより、スタックスマッシング攻撃の有無を確認する。スタックスマッシング攻撃においては、ローカル変数への代入操作を行い、境界チェックが不完全な場合に、そのローカル変数より上位のスタック領域を破壊する。したがって、プレヴィアス・フレームポインタと配列との間にガード変数を挿入されたスタックに対して、スタックスマッシング攻撃が行われると、関数の出口でガード変数が破壊されていることが

検出される。この時点でプログラムの実行を止めることにより、当該関数の呼び出し元へのリターンアドレス及びプレヴィアス・フレームポインタを保護することができる。

【0036】図1は、本実施の形態におけるスタック保護システムの全体構成を説明する図である。図1において、符号111はソース形式のプログラムである。符号112はスタック保護指令作成部であり、ソース形式のプログラム111における所定の箇所にガード変数を格納するための指令を作成してプログラムに付加する。符号121は、ガード変数を格納するための処理指令とガード変数の格納位置とを付加されたプログラムである。符号131はメモリパターンであり、コンピュータのメモリ装置におけるメモリパターンを形成する。符号132はスタック保護処理実行部であり、プログラム121を実行すると共に、その際にプログラム121に付加されている処理指令及びガード変数の格納位置に基づいてスタックの保護処理を実行する。符号133は保護数値であり、ガード変数として用いられる値である。

【0037】図2は、コンピュータにおいて、本実施の形態によりスタックの保護処理が行われている場合におけるメモリ装置のスタックの状態（メモリパターン）を示す図である。図2に示すように、スタックトップ（図では下側）から順に、ローカル変数領域210、プレヴィアス・フレームポインタ（PFP）220、及び当該呼び出し元へのリターンアドレス（RA）230が積まれている。そして、ローカル変数領域210における配列211の前、すなわち、配列211とプレヴィアス・フレームポインタ（PFP）220との間にガード変数（G）212が格納されている。ガード変数（G）212の位置は、配列211とプレヴィアス・フレームポインタ（PFP）220との間であればよく、この位置に格納される他のローカル変数との前後関係は問わない。

【0038】図3は、図1に示したスタック保護システムにおけるスタック保護指令作成部112の処理を説明するフローチャートである。スタック保護指令作成部112は、ソース形式のプログラム111のサブルーチンごとに、図3に示す処理を実行する。図3を参照すると、スタック保護指令作成部112は、まず着目中のサブルーチンにおいて、ローカル変数として配列が使用されているかどうかを確認する（ステップ301）。そして、配列が使用されている場合は、ガード変数（G）の格納位置を決定する（ステップ302）。ここで、ガード変数（G）の格納位置は、上述したように、プレヴィアス・フレームポインタと配列との間の任意の位置である。次に、当該サブルーチンを実行する際に、スタックにおける、ステップ302で決定された当該ガード変数（G）の格納位置にガード変数（G）を格納する旨の処理指令を生成する（ステップ303）。以上のようし

て、ガード変数を格納するための処理指令とガード変数の格納位置とを付加されたプログラム121が出力される。

【00039】プログラム121には、実際には、ガード変数を挿入すべきサブルーチン（関数）ごとに、ガード変数の宣言、関数の入口におけるガード変数の値の設定、関数の出口におけるガード変数の値の確認及び異常終了を実行するための命令が埋め込まれることとなる。このプログラム121の具体的な構成例は後述する。

【00040】図4は、スタック保護処理実行部132の処理を説明するフローチャートである。スタック保護処理実行部132は、スタック保護指令作成部112によりガード変数を格納するための処理指令とガード変数の格納位置とを付加されたプログラム121上のサブルーチンごとに、図4に示す処理を実行する。図4を参照すると、まずプログラム121の着目したサブルーチンにおいて、スタック保護指令作成部112により付加された処理指令があるかどうかを調べる（ステップ401）。

そして、処理指令がある場合は、当該プログラム121に処理指令と共に付加されているガード変数（G）格納位置に対応するスタック131上の位置にガード変数（G）の格納領域を生成する（ステップ402）。上述したように、ガード変数（G）の格納領域は、スタック131上のローカル変数領域であって、配列よりも前の位置に生成される。

【00041】次に、ステップ402で生成されたガード変数（G）の格納領域に、保護数値133を格納する（ステップ403）。これにより、スタック131は、図2に示した状態となる。ガード変数（G）として格納される保護数値133の内容は、特に制約はないが、攻撃者が推定できない値や攻撃によって格納することのできない値とする。そのような値の例として乱数を利用することができる。

【00042】次に、当該サブルーチンの処理（関数）を実行する（ステップ404）。そして、関数の出口でガード変数（G）として格納した値が保持されているかどうかを確認する（ステップ405）。ガード変数（G）の値が保持されていれば、通常通り終了する。また、ガード変数（G）の値が保持されていない場合は、スタックスマッシング攻撃が行われたことを意味するので、攻撃があったことを示すメッセージ及びエラーログを出力してプログラムを終了する（ステップ406）。以上のようにして、スタックスマッシング攻撃があれば、リターン処理の直前で検出でき、コンピュータシステムへの侵入を防止できることとなる。

【00043】本実施の形態のスタック保護システムは、結果的にプログラムの実行段階において、メモリパターンとして、図2に示したように、スタック上のプレヴィアス・フレームポインタと配列との間にガード変数を配置できれば良い。したがって、本実施の形態によるスタ

ック保護システムを実現するにあたり、プログラミング言語の種類やシステムの形態に応じて、種々の態様を探ることができる。例えば、プログラミング言語がコンパイラ言語である場合は、スタック保護指令作成部112の機能をコンパイラに実装することができる。この場合、スタック保護指令作成部112により処理指令及びガード変数（G）の格納位置を付加されたプログラム121は、実行形式のプログラムとなる。コンパイラのコード生成時にガード変数を挿入するためのプログラムを埋め込むこととすれば、オーバーフローを引き起こす変数よりもスタック上で上位アドレスにガード変数の宣言を配置すること、及び全ての関数の出口にプログラムを挿入することが容易となる点で好ましい。

【00044】これに対し、プログラミング言語がインタプリタ言語である場合は、スタック保護指令作成部112により処理指令及びガード変数（G）の格納位置を付加されたプログラム121の段階では、未だソース形式のプログラムである。また、プログラミング言語がコンパイラ言語である場合であっても、コンパイラの前段にスタック保護指令作成部112を構成する処理装置を置き、ソースプログラムの段階でガード変数を挿入するためのプログラムを埋め込むようにしても良い。図10に示したC言語のソースプログラムに対してガード変数を挿入するためのプログラムを埋め込む例を図5及び図6を参照して説明する。この場合、スタック保護指令作成部112は、図5に示す3つのプログラム、図10のプログラムの変数宣言部、関数の入口、関数の出口にそれぞれ挿入する。各プログラムが挿入された状態が図6に示すプログラムである。図6を参照すると、変数宣言部である符号601の位置、関数の入口である符号602の位置、関数の出口である符号603の位置に、それぞれ図5に示した3つのプログラムが挿入されている。

【00045】このように、スタック保護システムを実現するには、プログラミング言語の種類やシステムの形態に応じて、種々の態様を探ることが可能である。上記の他にも、ソース形式のプログラムを作成する段階で、所望の関数に対して、手作業により図5に示したようなプログラムを書き込むことも可能である。この場合、スタック保護指令作成部112の処理が手作業にて行われることに相当する。したがって、作成されたプログラムは最初からプログラム121の状態であり、スタック保護システムとして、スタック保護処理実行部132による処理だけが実行されることとなる。

【00046】図7は、本実施の形態のスタック保護システムを適用したコンピュータシステムの構成例を説明する図である。図7に示すコンピュータシステム700は、データ処理を行うデータ処理装置710と、データ処理装置710による処理の際に用いられるメモリ装置720と、データ処理装置710を制御する制御プログラム731を格納した記憶装置730とを備える。CP

Uにて実現されるデータ処理装置710は、制御プログラム731の制御の下、スタック保護処理実行部132の機能を実現する。メモリ装置720は、スタック保護処理実行部132による関数の実行において、スタック721上に、図2に示したガード変数を含むメモリパターンを形成する。さらに、データ処理装置710のスタック保護処理実行部132は、メモリ装置720のスタック721上に図2に示したメモリパターンを形成する際に、ガード変数の値として、記憶装置730から読み出した保護数値133を挿入する。なお、コンピュータシステム700において、コンパイル言語が実行される場合は、制御プログラム731は、図1に示したプログラム121に相当する実行形式のプログラムとなる。また、記憶装置730に格納された制御プログラム731を、ガード変数を格納するための処理指令とガード変数の格納位置とを付加されたソース形式のプログラムとすることもできる。この場合、データ処理装置710は、コンパイラの機能を併せて備えることとなる。さらに、制御プログラム731を、ガード変数を格納するための処理指令とガード変数の格納位置とを付加する前のソース形式のプログラムとすることもできる。この場合、制御プログラム731は、図1に示したソース形式のプログラム111に相当し、データ処理装置710は、スタック保護指令作成部112の機能を更に備えることとなる。さらにまた、図7においては、メモリ装置720と記憶装置730とを別個の構成要素として記載したが、物理的には単一の記憶モジュールにて構成することができる。

【0047】次に、本実施の形態におけるオーバーフローの低減について説明する。変数の型宣言とその参照とが正しく対応しているプログラムにおいては、バッファオーバーフローが発生する変数は、サイズ情報を持たない配列である。C言語などで使用されるこのような配列は、サイズ情報がないうえ、コンパイラの型チェックだけではバッファオーバーフローを防ぐことができない。一方、配列以外の変数では、プログラミング手法として型変換に注意を払えば、バッファオーバーフローを防ぐことが可能である。したがって、上述したように本実施の形態では、図2に示したように、スタックのローカル変数領域における配列とプレフィクス・フレームポインタとの間にガード変数を格納し、ローカル変数として配列を持たない場合はガード変数を格納しないこととした。これにより、システム全体のオーバーヘッドの低減を図ることができる。

【0048】ここで、変数の型宣言とその参照とが正しく対応しているプログラムにおいて、テスト領域に配列を持っていない関数（AF関数と称す）では、ガード変数を用いたバッファオーバーフローのテストを行う必要がないことを証明する。図8は、スタックにおけるテスト領域を説明する図である。スタック上のガード変数を用いてバッファオーバーフローのテストを行う領域は、当該ガード変数とそれよりも下位に現れるガード変数との間の領域、すなわち、図8の斜線が示した領域である。この領域を関数のテスト領域801と呼ぶこととする。証明は次の2つのステップで行う。1. AF関数のテスト領域801が破壊されるのは、それ以降に呼ばれる関数のテスト領域801がバッファオーバーフローを起こした場合であり、またその場合だけであることを示す。2. AF関数のテスト領域801が破壊された場合は、AF関数に実行制御が戻らないことを示す。

【0049】まず、第1のステップについて説明する。AF関数のテスト領域801には配列は存在しない。したがって、テスト領域801が破壊されたとしても、当該テスト領域がバッファオーバーフローの発生源ではない。したがって、テスト領域801の破壊の原因は、スタック上の当該関数フレームの上に置かれたテスト領域801となる。すなわち、AF関数から呼び出された関数の中でバッファオーバーフローが生じ、スタックの上位アドレスの向きに破壊して、AF関数のテスト領域を壊したものである。

【0050】次に、第2のステップについて説明する。第1のステップにより、AF関数のテスト領域801の破壊原因は、それ以降に呼ばれる他の関数にある。また上述したように、変数の型宣言とその参照とが正しく対応しているプログラムにおいては、バッファオーバーフローが発生する変数は配列であるから、AF関数におけるテスト領域801の破壊の原因となった領域は、配列である。配列を定義した関数では、ガード変数を用いたバッファオーバーフローのテストが実行されている。そして、テスト領域801を破壊し、かつリターンアドレスを操作するような攻撃があった場合は、必ずガード変数が破壊される。そのため、当該関数のリターン処理でプログラムは停止する。したがって、AF関数まで実行制御が戻ることはない。

【0051】以上で、テスト領域801に配列を持っていないAF関数では、ガード変数を用いたバッファオーバーフローのテストを行う必要がないことが証明された。なお、本実施の形態では、配列一般に対してガード変数を用いたスタック保護を行うこととして説明したが、特に文字配列のみを対象としてガード変数を用いたスタック保護を行うこととしても良い。これは、スタックスマッシング攻撃の対象となる配列が主に文字配列だからである。さらにその理由は、文字配列が、プログラム外部の情報表現に使用すること、データサイズを確認せずに処理できるように終端記号を持つ表現であること、といった特徴を持つことにある。したがって、ガード変数を用いたスタック保護を行う関数を、ローカル変数として文字配列を持つものに限定し、文字配列を持たないもの及び整数配列のような文字配列以外の配列を持つものについては、ガード変数を格納しないことによ

り、システム全体のオーバーヘッドを更に低減させることができる。

【0052】 以上のように、本実施の形態では、関数のリターン処理において、スタックのプレヴィアス・フレームポインタと配列との間に格納されたガード変数の有効性を確認する。これにより、スタック上のバッファオーバーフローを原因とするスタックスマッシング攻撃に関しては、ネットワーク経由での攻撃及びローカルユーザからの攻撃のいずれに対しても防止することができる。

【0053】 また、本実施の形態では、ローカル変数として配列を持つ関数のみを対象としてガード変数を用いたスタック保護を実行するため、従来技術において説明するように、全ての関数に対してスタック保護のための処理を実行する場合と比べてオーバーヘッドの削減を図ることができる。特にリターンアドレスの格納箇所を書き込み禁止にする場合や、リターンアドレス専用のスタックを用意する場合と比べると、大幅なオーバーヘッドの削減を実現できる。さらに、ローカル変数として文字配列を持つ関数のみを対象としてガード変数を用いたスタック保護を行うこととすれば、さらなるオーバーヘッドの削減を図ることができる。

【0054】 なお、本実施の形態では、上述のようにガード変数をローカル変数領域に格納するため、スタックのフレーム構造を変更することがない。したがって、プログラムに対するデバuggによるデバugg支援処理を行うことができなくなると行った不都合もない。

【0055】

【発明の効果】 以上説明したように、本発明によれば、スタック上のバッファオーバーフローを原因とするスタックスマッシング攻撃を完全に防止することができる。また、スタックスマッシング攻撃の防止手段を実装したことによる実行オーバーヘッドを削減することができる。

【図面の簡単な説明】

【図5】

```

● 変数宣言
volatile int guard;

● 関数入口
gv = guard_value;

● 関数出口
if (gv != guard_value){
    /*output error log */
    /*halt execution */
}

void foo()
{
    volatile int guard; ← 601
    char buf[128];

    gv = guard_value; ← 602
    strcpy (buf, getenv ("HOME"));

    if (gv != guard_value){
        /*output error log */ ← 603
        /*halt execution */
    }
}

```

【図6】

【図1】 本実施の形態におけるスタック保護システムの全体構成を説明するための図である。

【図2】 本実施の形態によりスタックの保護処理が実行されている場合のスタックの状態（メモリパターン）を示す図である。

【図3】 本実施の形態におけるスタック保護指令作成部の処理を説明するフローチャートである。

【図4】 本実施の形態におけるスタック保護処理実行部の処理を説明するフローチャートである。

【図5】 本実施の形態においてスタック保護の対象とすべきプログラムに挿入する、スタック保護を実行するためのプログラムの例を示す図である。

【図6】 図10のプログラムに対して図5のプログラムを挿入した状態を示す図である。

【図7】 本実施の形態のスタック保護システムを適用したコンピュータシステムの構成例を説明する図である。

【図8】 スタックにおけるテスト領域を説明する図である。

【図9】 C言語における関数呼び出し後のスタック状態を説明する図である。

【図10】 C言語で記述されたプログラムの例を示す図である。

【図11】 図10のプログラムに対してスタックスマッシング攻撃が行われた場合のスタックの状態を説明する図である。

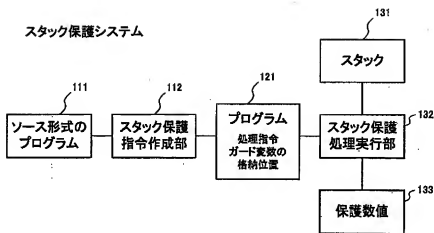
【符号の説明】

111…ソース形式のプログラム、121…スタック保護指令作成部、121…プログラム、131…スタック、132…スタック保護処理実行部、133…保護数値、700…コンピュータシステム、710…データ処理装置、720…メモリ装置、721…スタック、730…記憶装置、731…制御プログラム、801…テスト領域

【図9】



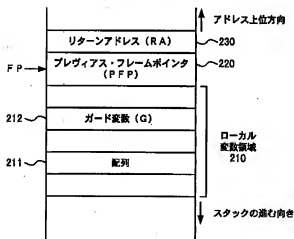
【図 1】



【図 2】

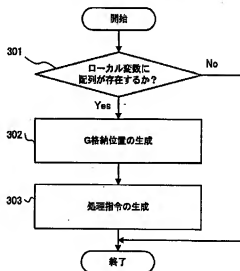
【図 3】

ガード変数を格納した場合の
メモリパターン



【図 10】

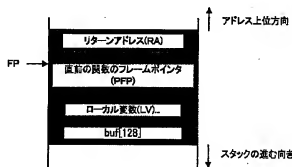
スタック保護指令作成部112の処理



【図 11】

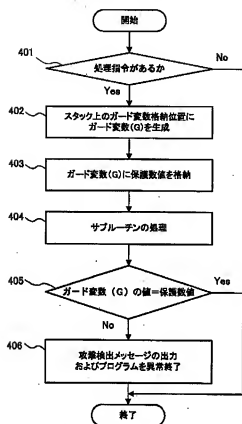
```

void foo()
{
    char buf[128];
    strcpy(buf, getenv("HOME"));
    ...
}
  
```

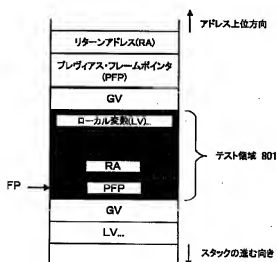


【図 4】

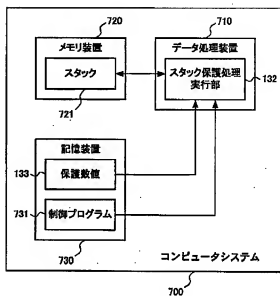
スタック保護処理実行部132の処理



【図 8】



【図 7】



フロントページの続き

(72)発明者 江藤 博明
神奈川県大和市下鶴間1623番地14 日本ア
イ・ビー・エム株式会社 東京基礎研究所
内

(72)発明者 依田 邦和
神奈川県大和市下鶴間1623番地14 日本ア
イ・ビー・エム株式会社 東京基礎研究所
内

Fターム(参考) 5B033 AA10 DE05 DE07 FA01 FA27